First person action networking game using virtual reality goggles in the Unity game engine

**Praca inżynierska   
 na kierunku Informatyka**

**Pierwszoosobowa sieciowa gra akcji dla gogli wirtualnej rzeczywistości**

**w silniku Unity**

**Kraków 2019**

**Politechnika Krakowska  
im. Tadeusza Kościuszki**

Wydział Fizyki, Matematyki i Informatyki

**Dawid Lubera**

Numer albumu: 117597

Praca wykonana pod kierunkiem:  
**dr inż. arch. Paweł Ozimek**

**Uzgodniona ocena:**......................................

...........................................................................podpisy promotora i recenzenta





SPIS TREŚCI

[1. WSTĘP 4](#_Toc535507818)

[1.1. CEL PRACY 4](#_Toc535507819)

[1.2. ZAKRES PRACY 4](#_Toc535507820)

[2. WPROWADZENIE TEORETYCZNE 5](#_Toc535507821)

[2.1. TECHNOLOGIE 5](#_Toc535507822)

[2.1.1. Unity 5](#_Toc535507823)

[2.1.2. Oculus Rift 6](#_Toc535507824)

[2.1.3. OpenVR 8](#_Toc535507825)

[2.1.4. Środowisko programistyczne 9](#_Toc535507826)

[2.2. KOMUNIKACJA W SIECI 12](#_Toc535507827)

[2.3. TECHNOLOGIE SIECIOWE W UNITY 13](#_Toc535507828)

[2.3.1. High Level API 13](#_Toc535507829)

[2.3.2. Low Level API 14](#_Toc535507830)

[2.3.3. Photon Unity Networking 14](#_Toc535507831)

[2.4. TECHNOLOGIE WIRTUALNEJ RZECZYWISTOŚCI 14](#_Toc535507832)

[2.4.1. Śledzenie pozycji 14](#_Toc535507833)

[2.4.2. Śledzenie ruchów głowy 16](#_Toc535507834)

[2.4.3. Śledzenie ruchów dłoni 16](#_Toc535507835)

[3. TWORZENIE APLIKACJI 18](#_Toc535507836)

[3.1. SPECYFIKACJA GRY 18](#_Toc535507837)

[3.2. PROJEKT 18](#_Toc535507838)

[3.2. MAPA 22](#_Toc535507839)

[3.3. KONFIGURACJA RUCHU DŁONI 25](#_Toc535507840)

[3.4. PORUSZANIE POSTACI 27](#_Toc535507841)

[3.4.1. Przemieszczanie modelu w przestrzeni 27](#_Toc535507842)

[3.4.2. Odwrotna kinematyka postaci 31](#_Toc535507843)

[3.5. KOMUNIKACJA SIECIOWA 36](#_Toc535507844)

[3.5.1. Wybór API 36](#_Toc535507845)

[3.5.2. Synchronizacja ruchu postaci 36](#_Toc535507846)

[3.6. STAN GRY 39](#_Toc535507847)

[4. TESTOWANIE 42](#_Toc535507848)

[5. PODSUMOWANIE 43](#_Toc535507849)

[BIBLIOGRAFIA 44](#_Toc535507856)

[SPIS ILUSTRACJI 45](#_Toc535507857)

# WSTĘP

## 1.1. CEL PRACY

Celem pracy jest stworzenie gry w silniku Unity. Aplikacja musi być stworzona z myślą o goglach wirtualnej rzeczywistości. Jednocześnie powinna używać technologii sieciowych do przesyłania położenia użytkownika. Celem gry jest wygranie w pojedynku bokserskim z drugim użytkownikiem. Obaj użytkownicy będą połączeni z serwerem i stan gry oraz położenia obu graczy w przestrzeni powinien być zsynchronizowany.

## 1.2. ZAKRES PRACY

W pracy znajduje się opis wszystkich używanych w procesie tworzenia technologii, opis implementacji poszczególnych komponentów aplikacji, sposoby testowania aplikacji, wnioski, które zostały wyciągnięte podczas pracy nad aplikacją oraz pomysły na rozwój aplikacji.

Rozdział II zawiera opis postawionego problemu komunikacji w sieci oraz opis technologii pozwalających na rozwiązanie i implementację postawionego zagadnienia. W rozdziale III zostały przedstawione szczegóły implementacji rozwiązania. Rozdział IV prezentuje sposoby testowania aplikacji. Ostatni, V rozdział stanowi podsumowanie wykonanej pracy.

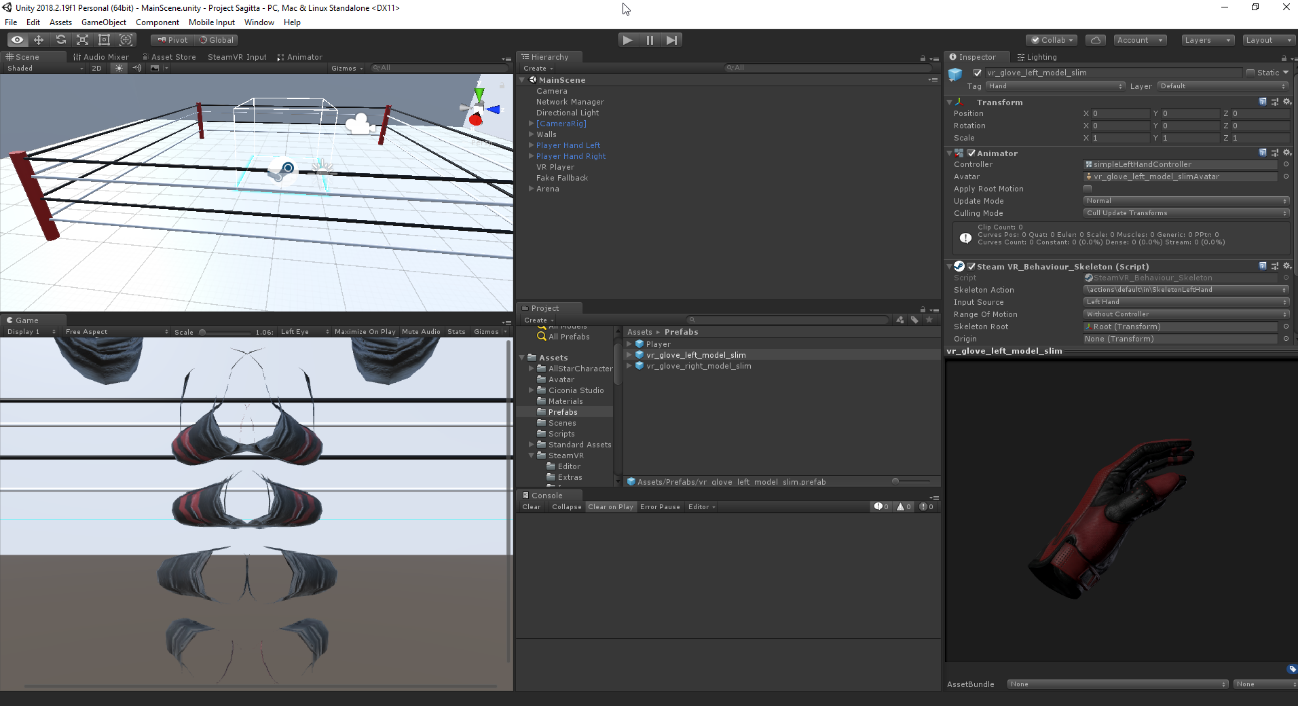
# WPROWADZENIE TEORETYCZNE

## TECHNOLOGIE

### 2.1.1. Unity

Unity jest rozbudowanym silnikiem do gier, który pozwala na tworzenie, wyświetlanie i zarządzanie dwu i trójwymiarowymi światami. Darmowa wersja programu pozwala na korzystanie ze znacznej większości funkcji. Ograniczenia wiążą się z brakiem wsparcia producenta, ilości graczy w rozgrywkach dla wielu graczy oraz maksymalny dochód ze sprzedaży, po którym wymagana będzie płatna licencja. Środowisko to pozwala na kompilację gry na wiele różnych platform w tym: Windows, Linux, Mac OS X, PS4, Xbox One, Nintendo Switch, Android oraz iOS.

W 2005 roku Unity zostało stworzone jako silnik do gier trójwymiarowych i wciąż jest to jego główne zadanie. Zawiera jednak wiele funkcji pozwalających w łatwy sposób tworzyć gry dwuwymiarowe. Pozwala także tworzyć gry i aplikacje dla gogli wirtualnej rzeczywistości. W przypadku niektórych gogli wymagane jest pobranie odpowiednich zasobów ze sklepu Unity – *Unity Asset Store[[1]](#footnote-1)*.



Rysunek 1. Główne okno edytora Unity, opracowanie własne

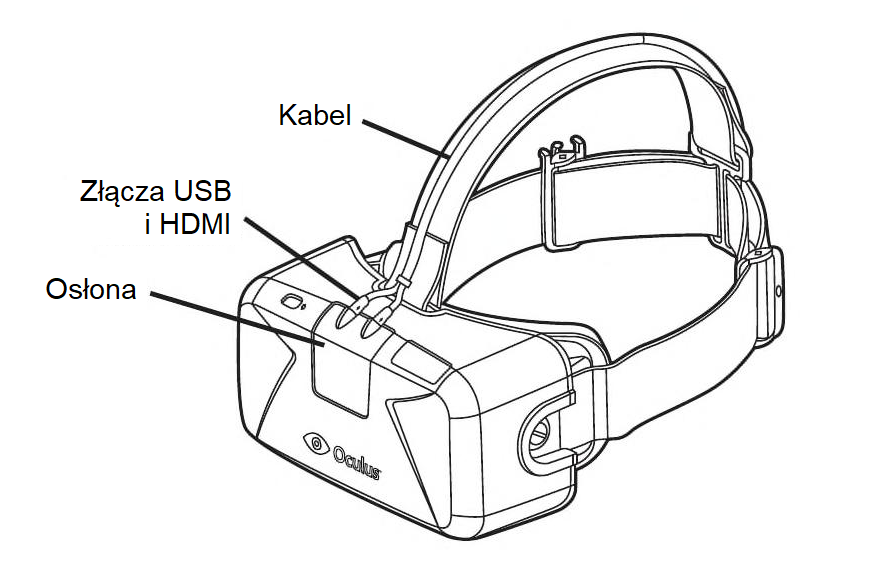
Edytor Unity pozwala na importowanie do gry modeli w wielu popularnych formatach w tym plików z programów: 3ds Max, Maya, Blender, SketchUp, a także popularny w wielu innych programach format OBJ. Umożliwia import tekstur  
i materiałów bezpośrednio z plików modeli lub osobno w postaci plików graficznym między innymi w formatach JPEG i PNG.

Pracując w Unity część zadań można wykonać bez programowania. Tworzenie świata gry, proste łączenia elementów lub animacji nie wymagają pisania kodu. Te czynności można wykonać poprzez przeciąganie odpowiednich elementów, wybór odpowiednich trybów pracy czy zaznaczanie pól wyboru. Przy programowaniu używany jest język C#, który został wybrany ze względu na możliwość używania na wielu platformach, szybki rozwój języka, niższy poziom skomplikowania składni niż C/C++  
i szybkość działania . Środowisko unity jest ciągle wspierane i rozwijane, dzięki czemu wprowadzane są nowe narzędzia i możliwości tworzenia aplikacji na wiele nowych urządzeń. Unity posiada także własny sklep, w którym można kupić gotowe skrypty, modele, pliki muzyczne czy narzędzia dodające nowe funkcjonalności do edytora.

### 2.1.2. Oculus Rift

Oculus Rift są pierwszymi goglami wirtualnej rzeczywistości, które odniosły komercyjny sukces. Pierwsza wersja deweloperska została wydana w marcu 2013 roku, a wersja konsumencka w styczniu 2016 roku. W grudniu tego samego roku zostały wydane kontrolery Touch, których opis znajduje się w dalszej części podrozdziału.

Gogle wirtualnej rzeczywistości pozwalają na obserwowanie wirtualnego świata w sposób symulujący obecność w tym świecie. Wyświetlają one dwa obrazy, po jednym dla każdego oka, które zasłaniają całe pole widzenia. Stosowane są odpowiednio zakrzywione soczewki oraz ruch kamery w grze lub filmie bazujący na faktycznych ruchach głowy. Sprawia to, że ludzki mózg faktycznie jest w stanie wczuć się w wydarzenia, które odbywają się na ekranach.



Rysunek 2. Schemat Oculus Rifta[[2]](#footnote-2)

Kontrolery Touch posiadają dokładne sensory pozwalające analizować pozycję, ruch i obrót dłoni użytkownika, a dokładne przyciski pozwalają odczytać informację o sile nacisku. Kontrolery pozwalają na dokładne symulowanie dłoni i odwzorowanie ruchu każdego palca. Gracz widząc ruchy swoich dłoni w grze bardziej angażuje się w rozgrywkę, immersja[[3]](#footnote-3) jest znacznie głębsza. Na każdym z kontrolerów znajdują się dwa zwykłe przyciski pozwalające odczytywać stany czy jest wciśnięty – *true,* czy nie – *false*, niewielki joystick kierunkowy, dotykowe miejsce obok przycisków, a także dwa triggery. Triggery w przeciwieństwie do przycisków pozwalają odczytać nie tylko informacje czy przycisk jest wciśnięty, czy nie, ale także siłę nacisku – wartość od 0 do 1. Jeden trigger jest umieszczony w pozycji palca wskazującego, a drugi, trzech palców: środkowego, serdecznego i małego. Każdy element zwraca także informację, czy użytkownik go dotyka, czy nie. Te wszystkie dane pozwalają stwierdzić w jaki sposób jest ułożona dłoń użytkownika.



Rysunek 3. Kontrolery Oculus Touch[[4]](#footnote-4)

### 2.1.3. OpenVR

OpenVR jest zestawem deweloperskim (SDK) stworzonym przez firmę Valve, aby ujednolicić tworzenie oprogramowania dla urządzeń wirtualnej rzeczywistości. Valve miało kluczowe znaczenie przy tworzeniu Oculus Rifta, a także stworzyło gogle HTC Vive i kontrolery dla dłoni Valve Knuckles. Wszystkie te urządzenia są wspierane przez OpenVR, a lista urządzeń, z których można używać z tym SDK jest znacznie dłuższa. Valve jest też twórcą dodatku do Unity dodającego wsparcie dla tego interfejsu.

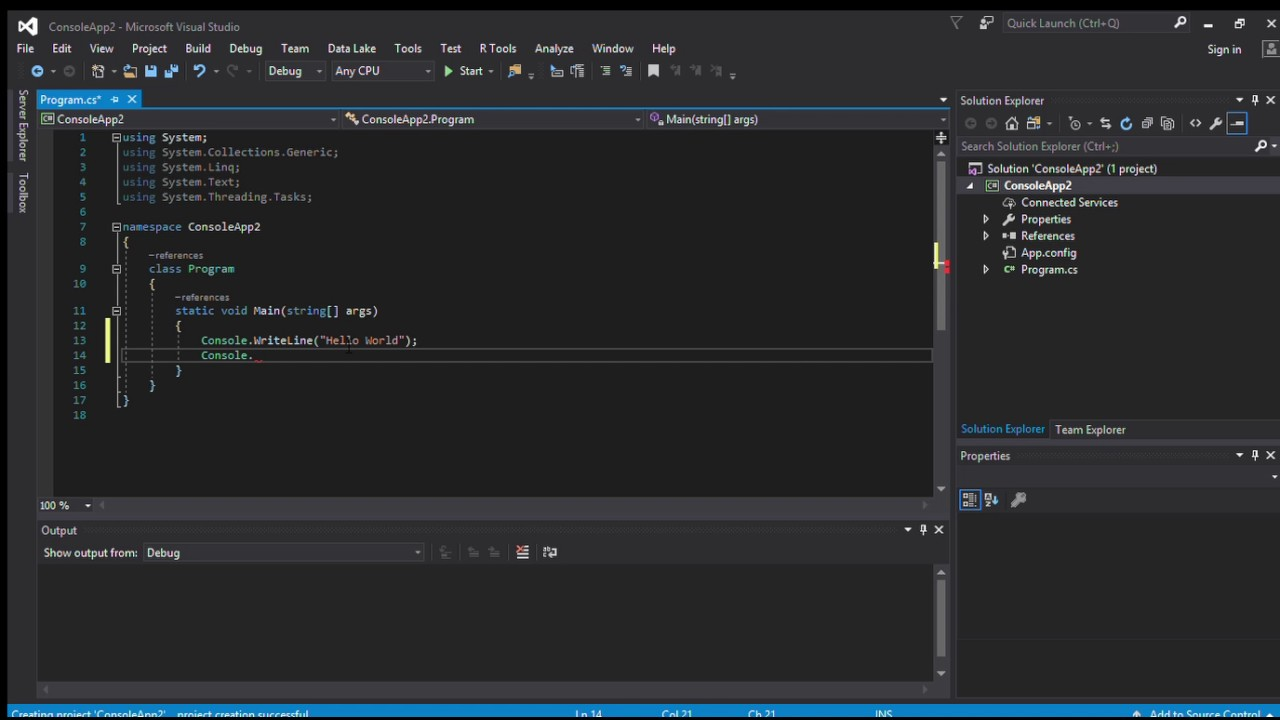


Rysunek 4. Okno przypisywania przycisków do akcji, opracowanie własne

Gry stworzone przy pomocy OpenVR pozwalają także na bardzo łatwą zmianę zastosowań danych przycisków zarówno przez deweloperów, jak i przez graczy. Pozwala także na analizę użycia przycisków w trakcie testowania aplikacji.

### 2.1.4. Środowisko programistyczne

Edytorem kodu, który został użyty w projekcie było Visual Studio. Głównym powodem takiego wyboru jest fakt, że jest on automatycznie instalowanym i najbardziej wspieranym przez Unity edytorem kodu na systemy Windows. Visual Studio jest zintegrowanym środowiskiem programistycznym stworzonym przez firmę Microsoft. Środowisko programistyczne pozwala na łatwiejsze zarządzanie kodem całego projektu. Visual Studio wspiera programowanie w wielu popularnych językach programowania i kodowania między innymi: C, C++, C#, JavaScript, CSS czy HTML. Visual Studio jest ciągle wspierane, a twórcy wprowadzają obsługę nowych funkcjonalności języków w kolejnych aktualizacjach.

****

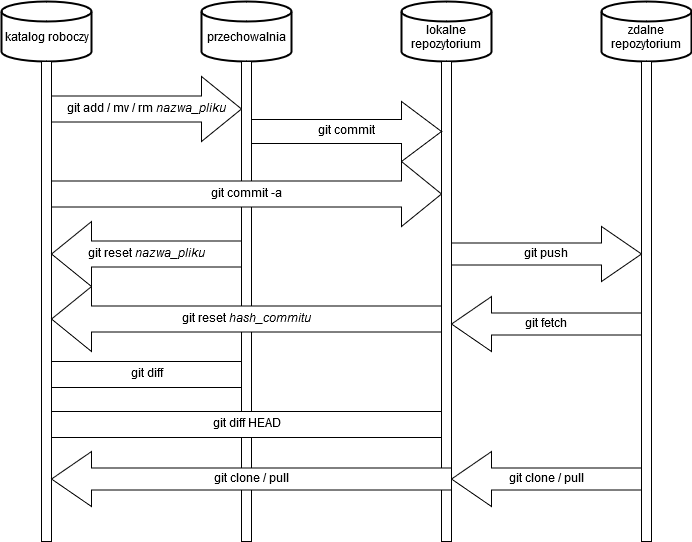
**Rysunek 5. Okno główne Visual Studio 2018, opracowanie własne**

Aktualnie jedynym językiem używanym przy programowaniu aplikacji w Unity jest C#. Jest to obiektowy język programowania stworzonym przez firmę Microsoft, którego pierwsza wersja ukazała się w lipcu 2000. Od tamtego czasu jest aktywnie rozwijany i wprowadzane są nowe funkcjonalności. Głównymi językami, na których bazuje C# są C++, Java i Delphi. Kod programu napisanego w tym języku jest kompilowany do specjalnego języka Common Intermediate Language. Taki program może zostać uruchomiony przez środowisko uruchomieniowe np. *.NET Framework*, *.NET Core* lub *Mono*.

System kontroli wersji jest narzędziem do śledzenia zmian w kodzie źródłowym. Narzędziem używanym w tym projekcie był Git. Pozwoliło to, przechwytywać historię zmian tworzonego projektu i zarządzać kodem projektu, co kilkukrotnie oszczędziło czas przy szukaniu powodu wystąpienia błędu w aplikacji. Pozwoliło także na łatwe przenoszenie kodu aplikacji pomiędzy różnymi komputerami, ponieważ kod źródłowy projektu przechowywany był w repozytorium na serwerze.

Podstawowa praca z systemem Git opiera się na używaniu kilku komend:

* *git add / git mv / git rm* – komendy pozwalające na dodawanie, przesuwanie i usuwanie plików w repozytorium
* *git commit* – zatwierdza dokonane zmiany i aktualizuje lokalne repozytorium
* *git push* – wysyła lokalne zmiany do repozytorium na serwerze
* *git fetch* – pobiera z aktualną wersję projektu i umieszcza w lokalnym repozytorium, nie modyfikuje plików projektu
* *git merge* – pozwala na scalenie plików, które były modyfikowane np. przez różne osoby w tym samym momencie
* *git pull* – pobiera najnowszą wersję projektu z i umieszcza w repozytorium jednocześnie scalając pliki
* *git reset* – w zależności od podanej wartości pozwala na usunięcie zmian z pliku lub pozwala na zresetowanie aktualnego repozytorium do konkretnych zmian
* *git clone* – pozwala na pobranie najnowszej wersji projektu i stworzenie lokalnego repozytorium



Rysunek 6. Schemat działania Git, opracowanie własne

## KOMUNIKACJA W SIECI

Sieć komputerowa to zespół logicznie ze sobą działających elementów. Fizycznymi składnikami sieci komputerowej są:

* Urządzenia sieciowe – są to między innymi komputery, telefony, serwery oraz sprzęt sieciowy (routery, przełączniki, koncentratory i tym podobne)
* Media transmisyjne – są to wszelkie media które łączą urządzenia sieciowe, media to zarówno połączenia przewodowe – okablowanie jak i bezprzewodowe – np. standardy 802.11, czyli WiFi

Protokół sieciowy to zestaw pewnych zasad według których wykonywana jest komunikacja w sieciach komputerowych. Implementacja danego protokołu w systemie każdego z połączonych urządzeń pozwala na komunikację w sieci.

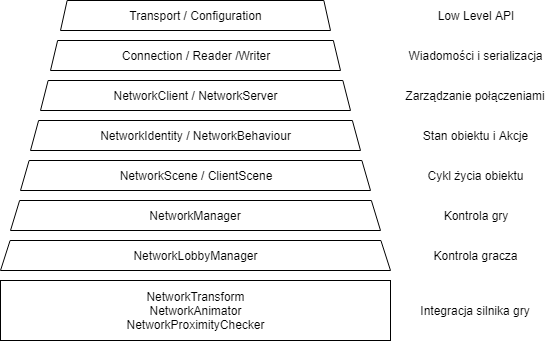
Komunikacja w sieci opiera się na modelu TCP / IP. Model ten opisuje całe zagadnienie komunikacji dzieląc je na mniejsze, współpracujące ze sobą warstwy. Wyróżniane są:

* Warstwa aplikacji – najwyższy poziom w którym działają aplikacje z których korzystają użytkownicy np. przeglądarka internetowa czy serwer WWW. W tej warstwie opisane są używane między innymi protokoły: HTTP, FTP, DNS, SMTP
* Warstwa transportowa – opiera się na wykorzystaniu portów określonych dla każdego połączenia, przez co kieruje właściwe informacje do właściwych aplikacji. Porty sieciowe umożliwiają identyfikację procesów w oddalonych od siebie systemach. Pozwala to na używanie wielu aplikacji korzystających z sieci na tym samym komputerze. W tej warstwie nawiązywane są połączenia z innymi komputerami
* Warstwa Internetu – ta warstwa wykorzystuje adresy IP do komunikacji. Tworzona jest droga którą musi przebyć pakiet, aby dotrzeć do drugiego komputera w sieci. W tej warstwie działają między innymi routery
* Warstwa fizyczna – jest najniższą warstwą i zajmuje się przekazywaniem danych przez fizyczne połączenia między urządzeniami sieciowymi

## TECHNOLOGIE SIECIOWE W UNITY

Unity posiada dwa wbudowane programistyczne interfejsy sieciowe oraz trzeci – *Photon* instalowany samodzielnie. Interfejs programistyczny pozwala na prostszą implementację rozwiązań sieciowych. Zapewnia funkcje, które w sposób łatwiejszy niż rozwiązania korzystające bezpośrednio z funkcji systemowych realizuje to samo. Dodatkowo interfejs sieciowy zapewnia implementację niezależną od systemu operacyjnego. Dwa wbudowane interfejsy sieciowe w Unity, z czego pierwszy z nich jest bazą dla drugiego to:

* Low Level API (LLAPI)
* High Level API (HLAPI)



Rysunek 7. Schemat warstwy sieciowej w Unity, opracowanie własne

Wybór interfejsu sieciowego został przedstawiony w rozdziale 3.5.1.

### 2.3.1. High Level API

High Level API (HLAPI) udostępnia logikę wymaganą w grach sieciowych bez konieczności wchodzenia w szczegóły przesyłania danych w protokołach sieciowych. Jest to możliwe dzięki korzystaniu z warstwy aplikacji modelu TCP / IP. HLAPI pozwala także na łatwe włączenie matchmakingu, czyli automatycznego parowania dwóch graczy, jeśli tylko są podłączeni do tego samego serwera. Darmowa wersja Unity pozwala na połączenie się z serwerami firmy Unity maksymalnie 20 graczy. W większości wypadków jest to bardzo wygodne i szybkie rozwiązanie. Ma ono jednak swoje wady, między innymi to, że wymusza, w jaki sposób klient komunikuje się z serwerem oraz jak wiele danych i w jakim formacie przesyła klient.

### 2.3.2. Low Level API

Low Level API (LLAPI) zostało zaprojektowane w taki sposób, aby zapewnić znacznie większą elastyczność. Kosztem elastyczności tego rozwiązania jest jednak ilość pracy, jaką należy wykonać do stworzenia systemu pozwalającego na połączenie i rozgrywkę dwóch klientów. LLAPI jest tylko nieco bardziej przystosowaną do obsługi gier nakładką na narzędzia działające w transportowej warstwie modelu TCP/IP.

### 2.3.3. Photon Unity Networking

Photon Unity Networking (PUN) jest dodatkowym assetem[[5]](#footnote-5) do Unity. Został zaprojektowany w podobny sposób co HLAPI. Podstawowa konfiguracja w projekcie wymaga jednak więcej pracy z uwagi na brak gotowego rozwiązania do interfejsu  
i łączenia użytkowników. Darmowa wersja pozwala na równoległą obsługę 20 użytkowników oraz ma ograniczenie na 500 wysłanych wiadomości na sekundę.

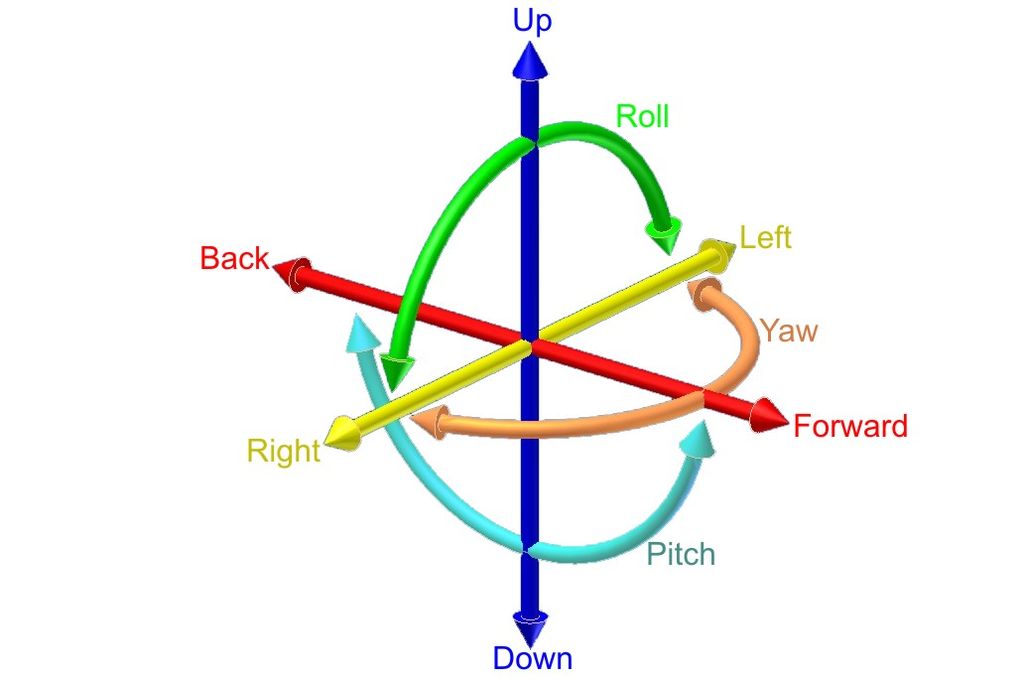
## TECHNOLOGIE WIRTUALNEJ RZECZYWISTOŚCI

### 2.4.1. Śledzenie pozycji

Śledzenie pozycji jest technologią, która pozwala urządzeniu na określenie swojej pozycji relatywnie do otoczenia. Jest to możliwe dzięki połączeniu odpowiedniego sprzętu z oprogramowaniem.

Pozwala na zmianę pola widzenia użytkownika w zależności od tego, czy porusza się, skacze, kuca czy pochyla do przodu. Z perspektywy użytkownika zwiększa połączenie świata fizycznego i wirtualnego. Pozwala także na użycie paralaksy[[6]](#footnote-6), która pomaga poinformować mózg o tym jak daleko w relacji do innych przedmiotów coś się znajduje.

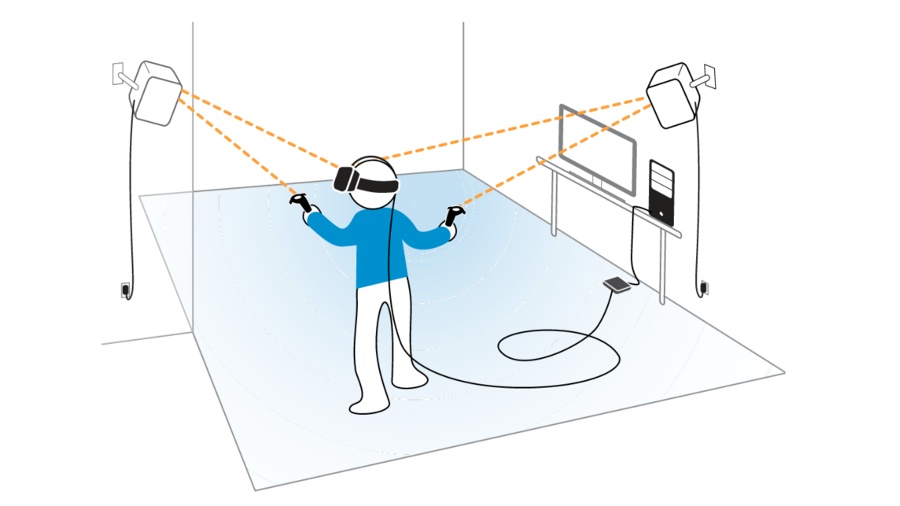
Warto zaznaczyć, że śledzenie pozycji nie jest tym samym co śledzenie ruchów głowy. To drugie pozwala tylko na śledzenie obrotów głowy, ale nie jej pozycji, ale obie technologie są wymagane, do zapewnienia tzw. sześciu stopni swobody (ang. *Six degrees of freedom, 6DoF*). Sześć stopni swobody reprezentują osie X, Y, Z oraz obroty wokół tych osi (ang. *pitch, yaw, roll*). Użycie takiego rozwiązania pomogło znacznie zredukować występowanie efektów choroby lokomocyjnej wśród użytkowników.



Rysunek 8. Sześć stopni swobody[[7]](#footnote-7)

Istnieje kilka różnych metod śledzenia pozycji. Wybranie odpowiedniej lub wystarczającej wymagało odpowiedzi na pytania jakiej dokładności i szybkości odświeżania, powierzchni użytkowej, mocy obliczeniowej potrzebuje urządzenie, czy będzie używane z komputerem, czy samodzielnie i w końcu znalezienie konsensusu między ceną a jakością.

Oculus Rift używa dwóch sensorów, które muszą być oddalone od siebie o co najmniej jeden metr. Dla wygody użytkownika wymagane jest, aby komputer pozwalał na renderowanie obu obrazów w 90 klatkach na sekundę.



Rysunek . Przykład śledzenia pozycji dla VR[[8]](#footnote-8)

### 2.4.2. Śledzenie ruchów głowy

Do śledzenia ruchów głowy używa się wieloosiowych żyroskopów. Rozwój smartfonów sprawił, że ta technologia jest bardzo tania i wydajna. Wiele modeli gogli korzysta ze światła podczerwonego do określania pozycji użytkownika. Tańsze gogle, które używają ekranu telefonu do wyświetlania obrazu nie obsługują tej technologii, co sprawia, że śledzenie ruchów ogranicza się właśnie do ruchu głowy. Droższe modele posiadają kilka żyroskopów, aby dokładniej liczyć rotację gogli.

Odczytywanie ruchu głowy jest drugą z kluczowych części przy sześciu stopniach swobody. Ruch ten jest zaimplementowany w podobny sposób do zmiany pozycji w wirtualnym świecie.

### 2.4.3. Śledzenie ruchów dłoni

Podobnie jak w przypadku śledzenia pozycji, śledzenie ruchów dłoni opisuje proces ciągłego przechwytywania pozycji dłoni użytkownika. Odczytane informacje pozwalają na poruszanie rękami i ciałem użytkownika w wirtualnym świecie. Używanie dłoni do interakcji z wirtualnym światem jest bardzo naturalną czynnością. Nawet użytkownicy gogli niewspierających śledzenia ruchów dłoni próbują chwytać przedmioty, mimo że bez tego nie są w stanie dokonać jakiejkolwiek interakcji. Kontrolery pozwalają na wprowadzenie dodatkowego realizmu do wirtualnego świata. Zwykle implementacja takiego rozwiązania opiera się na stworzeniu w wirtualnym świecie dłoni, które odwzorowują ruchy użytkownika.

Istnieją dwa główne sposoby implementacji ruchów dłoni:

* Pierwsza technika wymaga posiadania odpowiednich kontrolerów, po jednym dla każdej z dłoni, na których są przyciski, triggery czy touchapdy. Tę technikę wykorzystuje większość z popularnych gogli, gdyż jest ona obecnie najskuteczniejsza
* Druga technika uwzględnia zaawansowane algorytmy analizy obrazu z kilku kamer, umieszczonych zarówno na goglach, jak i w dodatkowych sensorach. Metoda ta jest jednak bardzo trudna do implementacji w wydajny i jednocześnie dokładny sposób, przez co obecnie jest rzadko wykorzystywana. Może się to jednak zmienić wraz z rozwojem technologii

Oculus Rift wykorzystuje pierwszą z tych technik w swoich kontrolerach Oculus Touch. Z tego powodu właśnie ta pierwsza technika została zaprezentowana w projekcie.

# TWORZENIE APLIKACJI

## 3.1. SPECYFIKACJA GRY

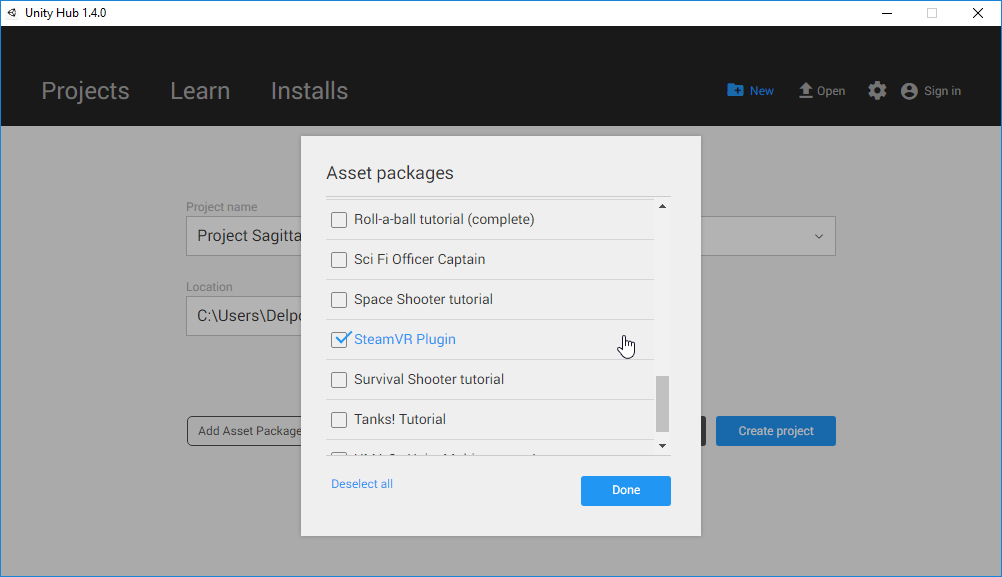
Gra pozwala na połączenie się gracza do serwera, do rozgrywki wymagane są jednak dwie osoby. W momencie gdy połączonych jest dwóch graczy, następuje faza rozgrywki w której celem każdego z użytkowników jest uderzenie głowy drugiego gracza. Uderzenie w głowę skutkuje zakończeniem rozgrywki i pojawieniem się napisu informującego o wygranej. U drugiego użytkownika pojawia się napis informujący o przegranej.

Projekt został zbudowany w oparciu o następującą strukturę:

* AllStarCharacterLibrary – model oraz dodatkowe animacje, tekstury i skrypty pobrane z assetem postaci do gry
* Materials – materiały stworzone dla projektu
* Prefabs – prefaby stworzone dla projektu
* Scenes – scena która jest używana do rozgrywki
* Scripts – skrypty, m.in. do poruszania postacią
* Standard Assets – podstawowe modele, animacje, materiały, które zostały dodane automatycznie przez Unity
* SteamVR, SteamVR\_Input – wszystkie skrypty biblioteki OpenVR

## 3.2. PROJEKT

Unity pozwala na łatwe stworzenie projektu klikając na przycisk *New* znajdujący się w prawym górnym rogu ekranu aplikacji startowej. W kolejnym kroku należy podać nazwę projektu, miejsce zapisu projektu w systemie plików oraz używany w projekcie szablon – 2D lub 3D. Okno to pozwala także na automatyczne dodanie dodatkowych assetów pobranych z *Unity Asset Store*.



Rysunek 10. Ekran tworzenia projektu w Unity, opracowanie własne

Konfiguracja projektu w Unity odbywa się według kryteriów:

* Nazwa projektu
* Szablon – 2D lub 3D
* Lokalizacja
* Dodatkowe zasoby

Tworząc projekt dla gogli wirtualnej rzeczywistości konieczne było użycie szablonu 3D oraz wybrania z menu dodatkowych zasobów *SteamVR Plugin*.

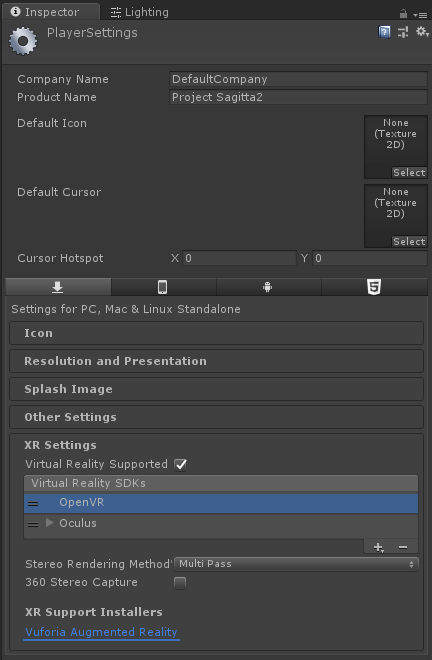
Po stworzeniu projektu przez Unity plugin SteamVR wyświetlił okno z zalecanymi ustawieniami konfiguracyjnymi projektu. Opcje, które powinny zostać zmienione to:

* Wyświetlanie okna rozdzielczości – zmiana z wyświetlania na ukrycie. W przypadku aplikacji stworzonych na VR wyświetlanie powinno być unikane, ponieważ obraz wyświetla się na ekranach w hełmie i wymaga odpowiedniej rozdzielczości.
* Umożliwienie zmiany rozmiaru okna podglądu – nie ma znaczenia dla wyświetlania gry na goglach
* Przestrzeń kolorów – zmiana z Gamma na Linear z uwagi na mniejsze obciążenie karty graficznej



Rysunek 11. Okno konfiguracji opcji pluginu SteamVR, opracowanie własne

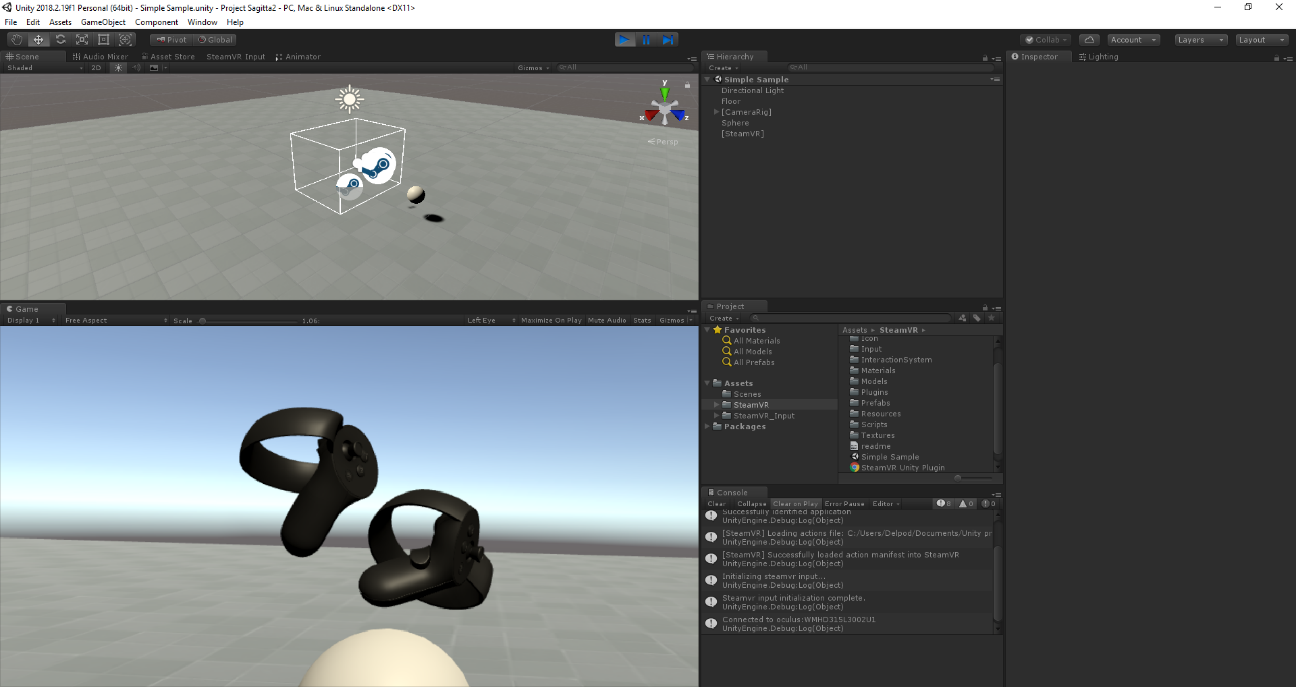
Kolejnym wymaganym krokiem jest zmiana ustawień projektu w Unity. Zostało to dokonane poprzez kliknięcie w oknie programu *Edit* > *Project Settings* > *Player*. W wyświetlanym oknie w sekcji XR zaznaczona została opcja *Virtual Reality Supported* oraz z uwagi na korzystanie z OpenVR zmieniona została kolejność SDK, tak, aby OpenVR znajdował się na pierwszym miejscu.



Rysunek 12. Konfiguracji Opcji XR, opracowanie własne

Do uruchomienia aplikacji wymagana była jeszcze scena, która zawiera kamerę oraz 2 obiekty odpowiadające każdej z dłoni. Plugin SteamVR zawiera taką podstawową scenę, więc została ona skopiowana do projektu i użyta jako baza do dalszej pracy.

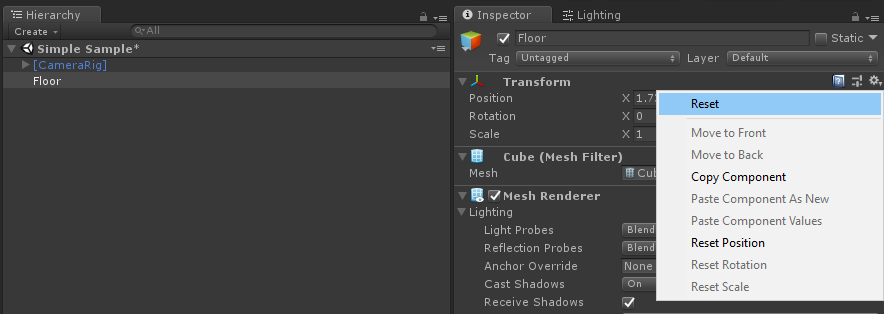
Uruchamiając projekt po raz pierwszy plugin SteamVR zaoferował możliwość konfiguracji wejścia. Podstawowe domyślne akcje zaproponowane przez plugin były wystarczające i wymagane było tylko ich wygenerowanie. Po ich zapisaniu można było już uruchomić projekt, który korzystając z gogli wyświetlał podstawową scenę oraz umożliwiał poruszanie kontrolerami.



Rysunek 13. Okno edytora z uruchomioną podstawową sceną, opracowanie własne

## 3.2. MAPA

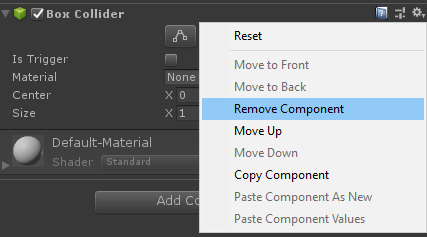
Mapa składa się z oświetlonego pokoju z ringiem na środku. Na początku tworzenia gry usunięta została zawartość sceny poza obiektem *[CameraRig]***.** Klikając prawym klawiszem myszy i wybierając *3D Object* > *Cube* w oknie hierarchii utworzony został nowy prostopadłościan. Następnie zresetowano transformację tego obiektu klikając w ikonę zębatki komponentu *Transform* i wybierając *Reset*.



Rysunek 14. Resetowanie komponentu w Edytorze, opracowanie własne

Odpowiednio ustawiając zmienne w komponencie *Transform* utworzona została podłoga o szerokości i długości 20 metrów. Obiekt stanowi podłogę pokoju. Aby utworzyć kolejne ściany pokoju podłoga została zduplikowana pięciokrotnie. Zduplikowanym obiektom nadane zostały odpowiednie transformacje, które sprawiały, że wszystkie 6 obiektów stworzyło prostopadłościan wyglądający jak pokój.

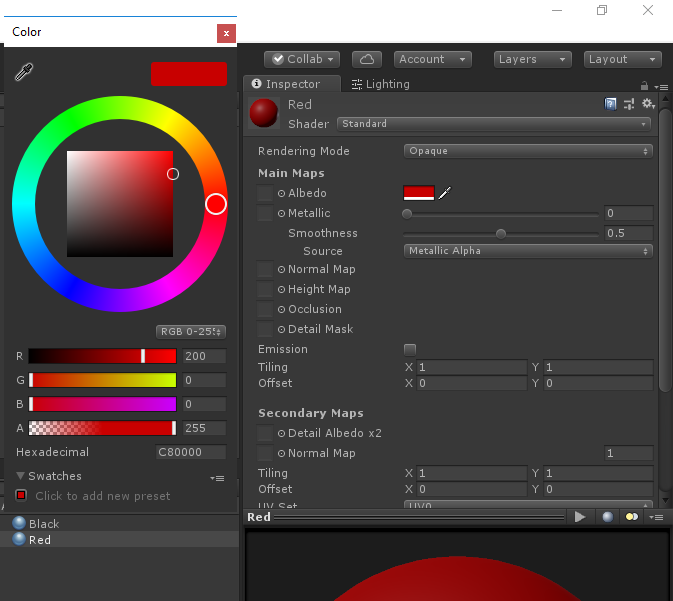
W odróżnieniu od podłogi ściany pokoju nie potrzebują wykrywania kolizji – użytkownik poruszać się będzie w obszarze mniejszym niż ściany. Ze wszystkich obiektów poza *Floor* usunięty został komponent *Box Collider*. W menu, które otwiera się po otworzeniu zębatki wybrana została opcja *Remove Component.*



Rysunek 15. Usuwanie komponentu Box Collider, opracowanie własne

Następnie w podobny sposób jak ściany, wybierając opcję *Create empty* utworzony został pusty obiekt, nazwany *Walls*. Obiekt ten także został zresetowany. Wszystkie utworzone ściany zostały przeniesione w hierarchii pod obiekt *Walls* stając się jego częścią.

Kolejną częścią było utworzenie ringu. Aby stworzyć słupki na rogach najpierw w menu otwierającym się po kliknięciu prawym klawiszem myszy należało wybrać opcję *Object 3D > Cyllinder.* Po utworzeniu został on nazwany *Pole1,* transformacja obiektu została zresetowana, a komponent *Capsule Collider* został z niego usunięty. Chcąc zmienić kolor obiektu konieczne było stworzenie nowego materiału. Materiał został utworzony klikając prawym klawiszem myszy w oknie *Project* i wybierając opcję *Create > Material*. Jedyną rzeczą, która musiała zostać zmieniona w nowym materiale jest kolor. Po kliknięciu w opcję *Albedo* otwiera się okno, które umożliwia zmianę koloru.



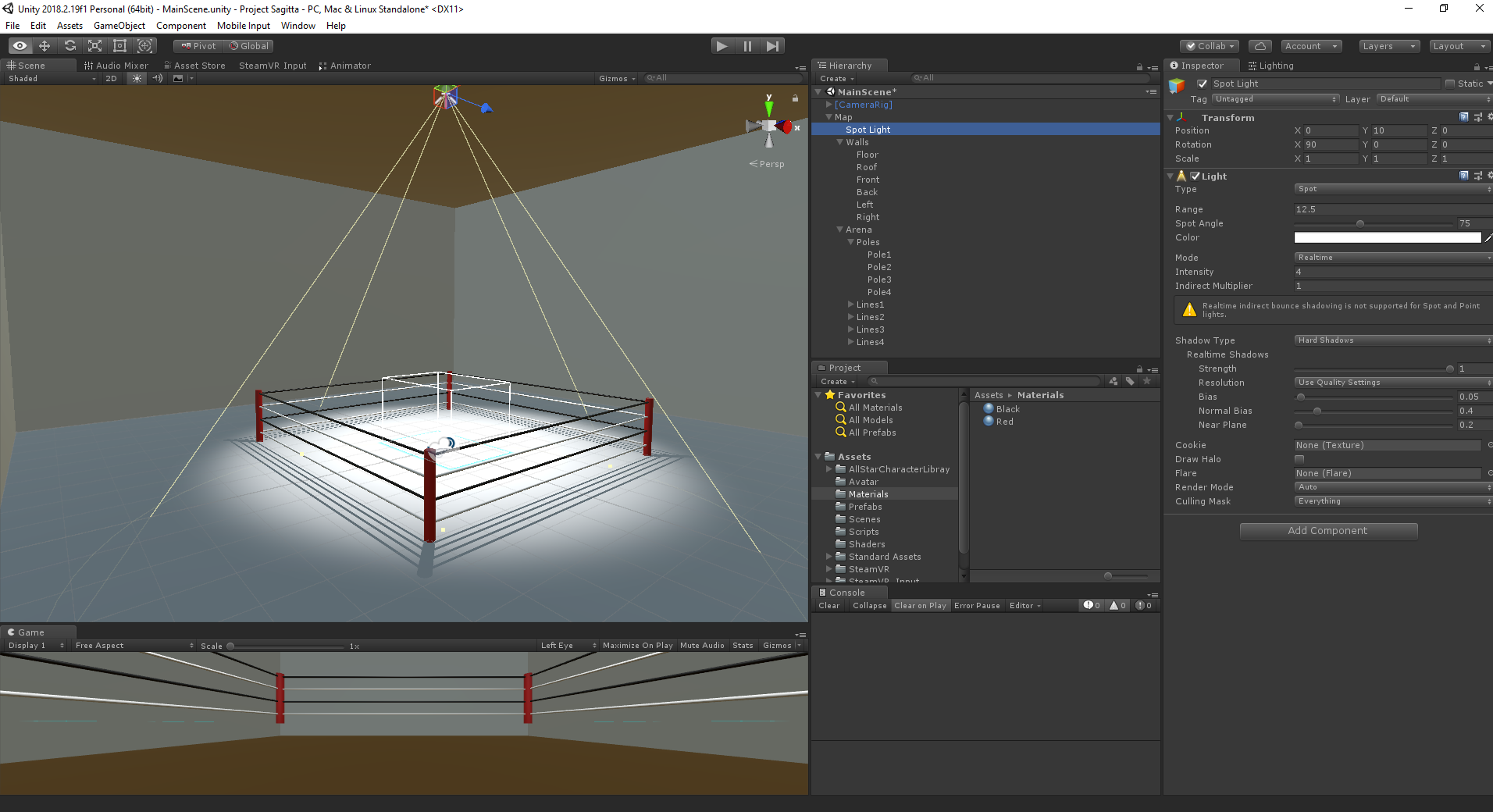
Rysunek 16. Tworzenie nowego materiału, opracowanie własne

Aby nadać kolor słupkowi należało przenieść materiał z okna *Project* na obiekt  
w scenie. Obiekt został trzykrotnie zduplikowany, a poszczególne słupki zostały ustawione w na planie. Podobnie jak ściany, obiekty zostały zgrupowane w nowym obiekcie o nazwie *Poles.*

Następnym krokiem było utworzenie lin, które znajdują się pomiędzy słupkami. Wartości ich transformacji zostały ustawione tak, by wszystkie znajdowały się między pierwszą parą słupków.

Stworzony został nowy, czarny materiał. Został on nadany obiektom *Line2*i *Line4.* Obiekty zostały zgrupowane do nowego obiektu o nazwie *Lines1.* Obiekt *Lines1* został następnie powielony trzykrotnie, a poszczególnym obiektom nadane zostały wartości transformacji, ustawiając je pomiędzy kolejnymi parami słupków. Obiekty *Poles, Lines1, Lines2, Lines3* i *Lines4* zostały zgrupowane w obiekcie *Map.*

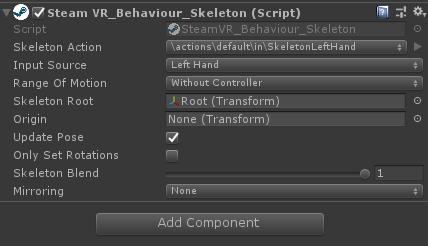
Następnie pod obiektem *Map* zostało utworzone światło typu *Spotlight* (światło skupiające wiązkę w ściśle określonym obszarze, światło sceniczne),obiekt został umieszczony na górze mapy, aby światło padało na arenę oświetlając ją, resztę pokoju pozostawiając w półmroku.



Rysunek 17. Wygląd utworzonej mapy wraz z pokazaną hierarchią obiektów, opracowanie własne

## 3.3. KONFIGURACJA RUCHU DŁONI

Do projektu zostały dodane modele dłoni, które znajdują się w folderze z plikami pluginu SteamVR. Nazwa modeli dłoni, które zostały użyte w projekcie to *vr\_glove\_left\_model\_slim* oraz *vr\_glove\_right\_model\_slim*. Do modeli dodany został komponent *Steam VR\_Behaviour\_Skeleton*. Wartości konfiguracyjne ustawione dla lewej dłoni w komponencie przedstawione są na rys. 18. Wartości dla prawej dłoni są bardzo podobne, różnice polegają w parametrach *Skeleton Action* i *Input Source*. Obie te opcje wskazują na wartości *RightHand*.



Rysunek 18. Konfiguracja komponentu szkieletu dla lewej dłoni, opracowanie własne

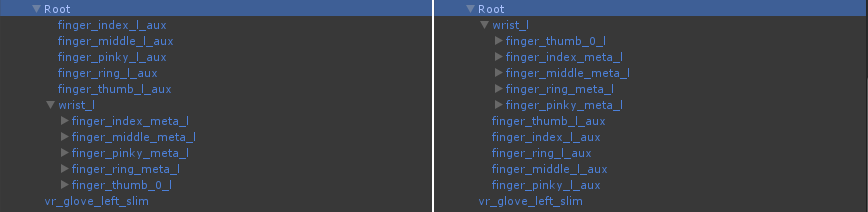
Wartości które można skonfigurować to:

* *Skeleton Action* – wybranie akcji, której ma używać skrypt
* *Input Source* – z którego fizycznego kontrolera ma pobierać wartości skrypt
* *Range Of Motion* – zakres ruchów modelu, porównanie dostępnych opcji znajduje się na rys. 19. Opcja *With Controller* jest bliższe aktualnemu układowi dłoni użytkownika, jednak nie pozwala na zamknięcie dłoni w grze, przez co odczucie immersji jest mniejsze
* *Skeleton Root* – nadrzędny obiekt dla całego systemu szkieletowego dłoni
* *Origin* – pozwala na określenie obiektu relatywnie do którego powinny poruszać się dłonie, przykładowym zastosowaniem jest np. poruszający się po mapie pojazd
* *Update Pose* – określa czy aktualizację pozycji i rotacji zajmować ma się ten skrypt czy będzie odpowiedzialny inny
* *Only Set Rotations* – opcja, która pozwala zmieniać samą rotację dłoni bez zmiany pozycji; działa tylko w połączeniu z *Update Pose*
* *Skeleton Blend* – oznacza szybkość zmiany pozycji dłoni – 1 to najszybciej, a 0 to brak zmiany pozycji
* *Mirroring* – pozwala na ustawienie lustrzane kopiowanie pozycji i rotacji kości dłoni



Rysunek 19. Efekty wyboru opcji Range of Motion: Without Controller (po lewej) i With Controller (po prawej), opracowanie własne

W dodanych modelach należało także zmienić kolejność punktów transformacji modelu. W oryginalnych rękawicach kości znajdują się w nieodpowiedniej kolejności  
i zgięcia nie odzwierciedlają faktycznego układu dłoni.



Rysunek 20. Nieprawidłowy (po lewej) i prawidłowy (po prawej) układ kości dłoni w modelu, opracowanie własne

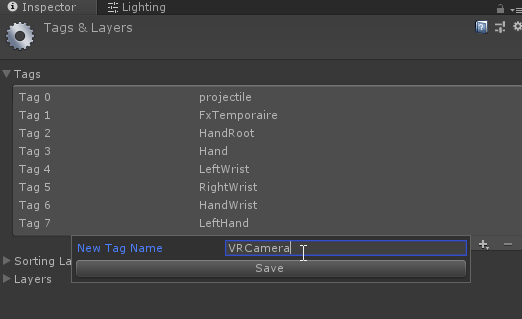
## 3.4. PORUSZANIE POSTACI

### 3.4.1. Przemieszczanie modelu w przestrzeni

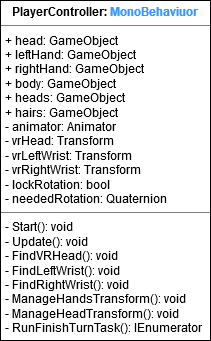
Stworzenie odpowiedniego przemieszczania w przestrzeni jest wymagane, aby w trakcie rozgrywki drugi użytkownik widział model gracza w sposób bliski temu,  
w którym miejscu i w jakiej pozycji aktualnie znajduje się użytkownik. Przy goglach wirtualnej rzeczywistości należy uwzględniać ruchy głowy oraz ruchy dłoni w przestrzeni świata.

Do projektu zaimportowany został model postaci pobrany z *Unity Asset Store[[9]](#footnote-9)*. Dla modelu utworzony został nowy skrypt o nazwie *PlayerController*. Skrypt posiada 3 zmienne publiczne, które następnie zostały ustawione w edytorze: *head, leftHand*i *rightHand.* W edytorze zmienne te reprezentowane są przez obiekty *Head, L Hand*i *R Hand,* które w strukturze znajdują się pod obiektem *ROOT* modelu. Używane są także zmienne *vrHead, vrLeftHand* i *vrRightHand.*

Przy komunikacji sieciowej, każdy z graczy będzie miał własne obiekty kontrolerów i głowy, przez co zmienne te muszą być zostać odnalezione dynamicznie. Aby to zrealizować utworzono etykiety dla modeli dłoni. W tym celu użyto komendy *Edit > Project Settings > Tags and Layers* w sekcji *Tags* dodane zostały etykiety *LeftWrist, RightWrist* i *VRCamera* (Rys. 21). Etykiety zostały nadane obiektom *Camera* w *[CameraRig]* oraz obiektom *wrist\_l*  i *wrist\_r* w *ROOT* modeli lewej i prawej dłoni.



Rysunek 21. Dodawanie etykiet do projektu, opracowanie własne



Rysunek 22. Schemat klasy PlayerController, opracowanie własne

Skrypt *PlayerController*, którego diagram prezentuje rys. 22 dziedziczy po klasie *MonoBehaviour*. Jest to wbudowana w Unity klasa, która jest klasą bazową dla wszystkich skryptów Unity. *MonoBehaviour* jest bardzo rozbudowaną klasą[[10]](#footnote-10), pozwala na dostęp do obiektu, w którym skrypt został użyty poprzez zmienną *gameObject* ale jej najważniejszymi funkcjami są te, opisujące czas życia projektu, są to między innymi:

* Awake() – uruchamiana po inicjalizacji wszystkich obiektów, nawet jeśli skrypt nie jest włączony
* Start() – uruchamiana po wykonaniu wszystkich funkcji Awake, jeśli skrypt i obiekt są włączone lub po pierwszym włączeniu skryptu
* Update() – przed rysowaniem każdej klatki
* OnDisable() – przed wyłączeniem skryptu
* OnEnable() – po włączeniu skryptu, tylko jeśli wcześniej była wykonana funkcja onDisable

W skrypcie *PlayerController*, którego diagram prezentuje Rysunek 22 utworzone zostały funkcje odpowiedzialne za odszukanie obiektów z odpowiednimi etykietami.

private void FindVRHead() {

GameObject head = GameObject.FindWithTag("VRCamera");

if (head) {

vrHead = head.transform;

}

}

Listing 1. Szukanie obiektu kamery VR

W listingu 1 szukany jest obiekt z etykietą *VRCamera* i przypisywany do zmiennej. Podobne funkcje odpowiadają za szukanie lewego oraz prawego modelu dłoni.

Aby zmieniać pozycję dłoni wymagane było przepisanie wartości z pozycji oraz rotacji dłoni w świecie. Należy jednak zauważyć, że z uwagi na to, jak skonstruowany został model postaci potrzebna była dodatkowa rotacja. Kod odpowiadający za ruch znajduje się w listingu 2.

leftHand.transform.position = vrLeftWrist.position;

leftHand.transform.rotation = vrLeftWrist.rotation;

leftHand.transform.Rotate(90f, 0f, -90f);

rightHand.transform.position = vrRightWrist.position;

rightHand.transform.rotation = vrRightWrist.rotation;

rightHand.transform.Rotate(-90f, 180f, -90f);

Listing 2. Ustawianie i przekształcanie pozycji i transformacji obiektów

Rotacja w unity przedstawiana jest za pomocą zmiennych typu *Quaternion[[11]](#footnote-11),[[12]](#footnote-12)*, aby uprościć obliczenia rotacji dla komputera. Aby jednak ułatwić pracę programistom, kwaterniony są przekształcane na kąty Eulera. Przekształcanie to przydaje się w wielu momentach. Jednym z nich była chęć uniemożliwienia rotacji modelu w sytuacjach, które sprawiałyby, że model wyglądałby źle, np. model głowy skierowany byłby do środka ciała. Do ograniczenia rotacji posłużył kod poniżej, który blokuje za duże obroty głowy względem ciała na osiach X i Z.

transform.position = new Vector3(

vrHead.position.x,

transform.position.y,

vrHead.position.z);

if (head.transform.rotation.eulerAngles.x > 60f && head.transform.rotation.eulerAngles.x < 180f) {

head.transform.rotation = Quaternion.Euler(

60f,

head.transform.rotation.eulerAngles.y,

head.transform.rotation.eulerAngles.z);

}

if (head.transform.rotation.eulerAngles.z > 300f) {

head.transform.rotation = Quaternion.Euler(

head.transform.rotation.eulerAngles.x,

head.transform.rotation.eulerAngles.y,

300f);

} else if (head.transform.rotation.eulerAngles.z < 235f) {

head.transform.rotation = Quaternion.Euler(

head.transform.rotation.eulerAngles.x,

head.transform.rotation.eulerAngles.y,

235f);

}

Listing 3. Ustawianie pozycji głowy oraz ograniczanie nieprawidłowych rotacji

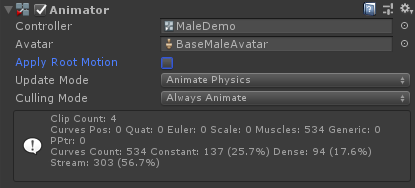
Listing 3 przedstawia kod używany do zmiany pozycji głowy. Położenie głowy na osi Y nie jest zmieniane. Skrypt generuje też nowe rotacje, jeśli te, które aktualnie są używane nie wyglądają poprawnie. Wartości które są nieodpowiednie zostały wyznaczone eksperymentalnie. Do utworzenia rotacji typu *Quaternion* na podstawie kątów Eulera służy funkcja *Quaterion.Euler().*

### 3.4.2. Odwrotna kinematyka postaci

Kinematyka odwrotna pozwala na zmianę pozycji elementów będącymi rodzicami przez obiekty będące dziećmi. Przykładem jest tutaj ruch dłoni, który zmienia układ rąk i tułowia.

W projekcie przyjęte zostało, że jeśli wartość absolutna różnicy rotacji kamery sterowanej kontrolerem na głowie na osi Y oraz aktualnej rotacji modelu postaci wynosi 30 lub więcej stopni to należy wykonać obrót tej postaci o 45 stopni w stronę obrotu kamery. Po obrocie wartość absolutna różnicy wynosi około 15 stopni.

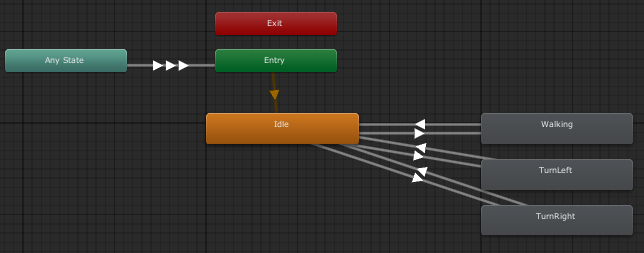
Aby obrót postaci był możliwy z animacjami w obiekcie postaci stworzony skonfigurowany został komponent *Animator*. Odpowiednie obiekty zostały pobrane i zaimportowane wraz z modelem. Konfiguracja przedstawiona jest na (Rys. 19).



Rysunek 23. Konfiguracja komponentu Animator, opracowanie własne

Następnie potrzebne było skonfigurowanie animacji. Okno *Animator* zostało uruchomione poprzez wybranie opcji *Window > Animation > Animator.* W karcie *Layers* przy *Base Layer* klikając ikonę zębatki włączono *IK Pass*, jest to opcja, która jeśli jest uruchomiona sprawia, że *Animator* oblicza odwrotną kinematykę*.* Na karcie *Parameters* zostały dodane trzy parametry: *AbsoluteMovement* typu *Float*, w którym jest zapisywane z jaką prędkością porusza się obiekt oraz *TurnLeft* i *TurnRight* typu *Bool*, gdzie zapisywane jest czy i w którą stronę obiekt powinien się obracać*.*

Kolejnym krokiem było stworzenie stanów i przejść między nimi. Cztery stany potrzebne do poruszania to *Idle* – stan gdy postać stoi*, Walking* – stan dla chodzenia*, TurnLeft –* stan gdy postać obraca się w lewo i *TurnRight –* stan gdy postać obraca się w prawo. *Idle* jest stanem domyślnym i będzie on wykonywany zawsze, gdy nie jest wykonywany żaden inny stan. Zostało to zrobione przez wybranie opcji *Set as Layer Default State*. Poprzez opcję *Make Transition* między stanem *Idle* a *Walking, TurnLeft i TurnRight* zostało utworzona możliwość zmiany stanu.

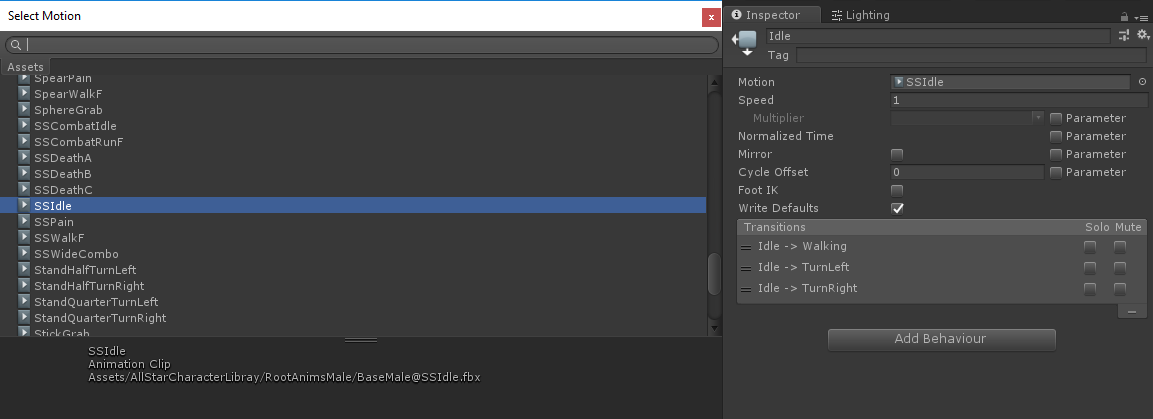


Rysunek 24. Wygląd maszyny stanów po dodaniu przejść między stanami, opracowanie własne

Przejście ze stanu *Idle* do *Walking* nie powinno następować przy niewielkich ruchach postaci, ale nie można też pozwolić, żeby użytkownik przemieszczał się całkowicie bez animacji. W sekcji *Conditions* wybrany został parametry *AbsoluteMovement Greater 0.03*. Natomiast przy przejściu ze stanu *Walking* do *Idle* ustawione zostały parametry *AbsoluteMovement Less 0.03.*Wartość 0.03 została wyznaczona doświadczalnie. W obu przejściach odznaczona została też opcja *Has Exit Time*. Opcja ta pozwala na przerwanie animacji jeśli warunek zmienił się przed jej ukończeniem.

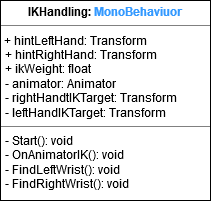
W przypadku stanów zmiany ze stanu *Idle* do *TurnLeft* i *TurnRight* w sekcji *Conditions* ustawione zostały parametry *TurnLeft true* i *TurnRight true*. Przy przejściu w drugą stronę warunek został zmieniony na *false.* Tutaj także wyłączona została opcja *Has Exit Time*.

Kolejnym krokiem było skonfigurowanie animacji dla poszczególnych stanów. Pierwszym z nich był stan *Idle*. Dla tego stanu wybrana została animacja *SSIdle* pobrana wraz z modelem. Animacja opisuje delikatne poruszanie się postaci w momencie gdy stoi. W stanach *Walking, TurnLeft* i *TurnRight* zostały odpowiednio wybrane animacjeo nazwach *SSWalkF* – animacja przedstawiająca chodzenie, *StandQuarterTurnLeft –* animacja niewielkiego obrotu w lewo i *StandQuarterTurnRight –* przedstawiająca obrót w prawo*.*



Rysunek 25. Wybór animacji dla stanu Idle, opracowanie własne

Po skonfigurowaniu animacji nastąpiła faza ich implementacji. Do obiektu postaci dodane zostały dwa puste obiekty, które zostały ustawione w miejscu, w które mniej więcej powinny być skierowane łokcie.



Rysunek 26. Diagram klasy IKHandling, opracowanie własne

Utworzony został nowy skrypt nazwie *IKHandling,* jego diagram został przedstawiony na Rysunku 26*.* Skrypt ten podobnie jak *PlayerController* znajduje obiekty z etykietami *LeftWrist* i *RightWrist*. Na ich podstawie układane są animacje. Kod odpowiadający za zmianę modelu w zależności od położenia dłoni znajduje w listingu 4.

animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, ikWeight);

animator.SetIKPosition(AvatarIKGoal.LeftHand, leftHandIKTarget.position);

animator.SetIKRotationWeight(AvatarIKGoal.LeftHand, ikWeight);

animator.SetIKRotation(AvatarIKGoal.LeftHand, leftHandIKTarget.rotation);

animator.SetIKHintPositionWeight(AvatarIKHint.LeftElbow, ikWeight);

animator.SetIKHintPosition(AvatarIKHint.LeftElbow, hintLeftHand.position);

Listing 4. Ustawianie opcji animatora

Skrypt ustawia pozycję, rotację i wskazanie miejsca skierowania łokci oraz wagę poszczególnych elementów. W projekcie wszystkie wagi zostały ustawione na 1. Obiekt klasy *IKHandling* odpowiada za ustawienie pozycji rąk w zależności od ruchu dłoni.

Kolejnym stworzonym elementem było uwzględnianie ruchów modelu przy ruchu głowy i obrotów w konkretną stronę. W *PlayerController* obliczana jest szybkość ruchu i ustawiany jest odpowiedni parametr animatora, co prezentuje listing 5.

float absoluteMovement = (Mathf.Abs(transform.position.x - vrHead.position.x) + Mathf.Abs(transform.position.z - vrHead.position.z)) \* Time.deltaTime \* 1000f;

animator.SetFloat("AbsoluteMovement", absoluteMovement);

Listing 5. Obliczanie wartości absolutnej ruchu postaci w przestrzeni

Następnie należało obliczyć różnicę kątów między rotacją głowy a rotacją postaci oraz jeśli rotacja ta jest większa niż 30 stopni to podjąć odpowiednią akcję, chyba że użytkownik jest w trakcie animacji. Podejmowane akcje to blokada animacji, zmiana kąta, który ma osiągnąć postać oraz uruchomienie odpowiedniej animacji. Jeśli użytkownik jest w trakcie animacji to powinna nastąpić rotacja postaci w tym kierunku. Implementację powyższych założeń przedstawia kod w listingu 6.

if (!lockRotation) {

float phi = Mathf.Abs(a - b) % 360f;

float distance = phi > 180f ? 360f - phi : phi;

float signedDistance = distance \* ((a - b >= 0 && a - b <= 180) || (a - b <= -180 && a - b >= -360) ? 1 : -1);

if (signedDistance > 30f) {

lockRotation = true;

StartCoroutine(RunFinishTurnTask());

neededRotation = Quaternion.Euler(0f, a - 45f, 0f);

animator.SetBool("TurnLeft", true);

} else if (signedDistance < -30f) {

lockRotation = true;

StartCoroutine(RunFinishTurnTask());

neededRotation = Quaternion.Euler(0f, a + 45f, 0f);

animator.SetBool("TurnRight", true);

}

} else {

transform.rotation = Quaternion.RotateTowards(transform.rotation, neededRotation, Time.deltaTime \* 180f);

}

Listing 6. Obliczanie kierunku obrotu i jego wykonanie



Rysunek 27. Wygląd postaci po zmianach w odwrotnej kinematyce, opracowanie własne

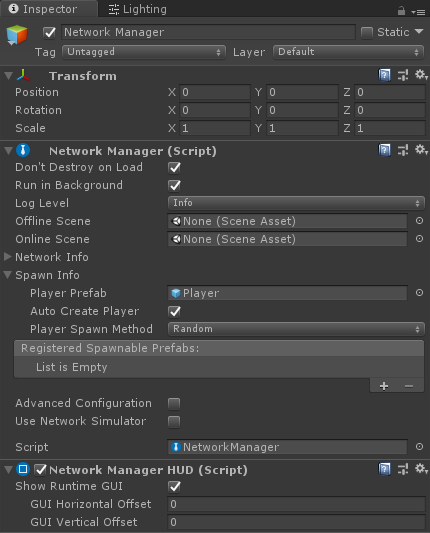
## 3.5. KOMUNIKACJA SIECIOWA

### 3.5.1. Wybór API

Przy wyborze API w projekcie najważniejszymi kryteriami była pełna obsługa synchronizacji wszystkich niezbędnych transformacji oraz niewielki nakład pracy. Z uwagi na chęć przesyłania dużej ilości informacji przy transformacjach kości dłoni i brak doświadczenia z Photon Unity Networking trudno było stwierdzić, czy 500 wiadomości na sekundę byłoby wystarczającą ilością. Z kolei Low Level API jest co prawda najbardziej elastycznym rozwiązaniem, ale ze względu na ogromną ilość pracy konieczną do wykonania rozwiązania, które byłoby wydajne, wybór padł na High Level API.

### 3.5.2. Synchronizacja ruchu postaci

Przed rozpoczęciem synchronizacji obiekt postaci należało zapisać jako prefab[[13]](#footnote-13). Aby synchronizacja przez Internet zadziałała utworzono nowy pusty obiekt, który został nazwany *Network Manager*. Do tego obiektu dodane zostały komponenty, skrypty *Network Manager* i *Network Manager HUD*. Pierwszy zapewnia komunikację sieciową, a drugi prosty interfejs pozwalający połączyć się z serwerem. W *Network Managerze* najważniejsze było ustawienie zmiennej *Player Prefab* na stworzony wcześniej prefab.

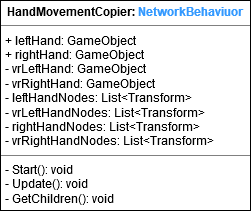


Rysunek 28. Konfiguracja Network Managera, opracowanie własne

Do stworzonego prefabu dodane teraz zostały komponenty *Network Identity*i *Network Transform*. W pierwszym z nich należy włączyć opcję *Local Player Authority*, a w drugimzmienić *Rotation Axis* na *XYZ (full 3D),* aby zmiany w ruchu obiektu były przekazywane niezależne od osi, na której występują. Aby transformacje obiektów niżej w hierarchii także były synchronizowane, należało utworzyć *Network Transform Child* dla każdego elementu, który miał być zsynchronizowany – dłoni, palców i głowy Aby animacje postaci były synchronizowane przez sieć dodano także *Network Animator* i zaznaczono wszystkie animacje.

W skryptach *PlayerController* i *IKHandling* wymagane są zmiany, ponieważ w obecnym stanie skrypty dla obiektów postaci będą się także uruchamiać na komputerach innych graczy. Należy sprawdzić, czy obiekt jest lokalnym obiektem gracza i tylko wtedy uruchomić.

Należy także zauważyć, że w tym momencie obiekty dłoni znajdowały się poza obiektem postaci. Nie jest jednak możliwe przeniesienie ich do tego obiektu, gdyż posiadają one skrypty *Steam VR\_Behaviour\_Skeleton*, które muszą być używane lokalnie, na komputerze użytkownika. Obiekty te zostały jednak skopiowane, a skrypt ten został usunięty. Brak tego skryptu nie pozwala jednak na ruchy dłoni. Dodatkowym skryptem, uruchamianym tylko lokalnie, który został do tego celu utworzony jest *HandMovementCopier*, którego diagram przedstawiony jest na Rysunku 29.



Rysunek 29. Diagram klasy HandMovementCopier, opracowanie własne

Pozwala on skopiować dane transformacji wszystkich dzieci obiektu na inny obiekt o identycznej strukturze. Obiekty dłoni, do których zostaną przekazane dane transformacji ustalone są w prefabie, a obiekty dłoni kontrolerów ładowane są dynamicznie. Kopiowanie pozycji i rotacji następuje przy każdej ich zmianie i jest przedstawione w kodzie poniżej.

for (int i = 0; i < leftHandNodes.Count; ++i) {

leftHandNodes[i].position = vrLeftHandNodes[i].position;

leftHandNodes[i].rotation = vrLeftHandNodes[i].rotation;

}

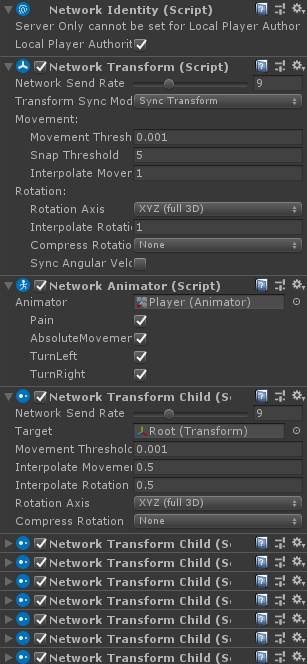
for (int i = 0; i < rightHandNodes.Count; ++i) {

rightHandNodes[i].position = vrRightHandNodes[i].position;

rightHandNodes[i].rotation = vrRightHandNodes[i].rotation;

}

Listing 7. Kopiowanie pozycji i rotacji dłoni sterowanej kontrolerem do kopii tej dłoni

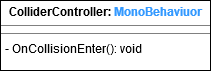


Rysunek 30. Konfiguracja skryptów sieciowych obiektu postaci, opracowanie własne

## 3.6. STAN GRY

Stan gry informuje użytkowników, czy rozgrywka wciąż trwa, czy została zakończona. Stan gry jest zmieniany na zakończony, gdy jeden z użytkowników zostanie uderzony przez drugiego. Aby możliwe wykrycie uderzeń potrzebny jest system kolizji dla obiektów dłoni i głowy prafabu *Player.*

Do obiektu lewej i prawej dłoni dodany został komponent typu *Capsule Collider,* który ustawiony został tak, aby pasować do wielkości dłoni*.* Dodany został też komponent *Rigidbody*, gdyż jest on wymagany do wykrywania kolizji w silniku Unity. *Rigidbody* sprawia, że obiekt staje się obiektem fizycznym. Domyślnie oznacza to, że będzie oddziaływał na siły innych ciał i na grawitacje. W przypadku dłoni gracza nie jest to pożądany efekt, dlatego opcja *Is Kinematic* została włączona. Po jej włączeniu na obiekt nie oddziałują już żadne siły, lecz wciąż jest uznawany za obiekt fizyczny. Aby kolizje między obiektami były wykrywane, gdy opcja *Is Kinematic* jest włączona,  
w ustawieniach fizyki dla projektu opcja *Contact Pairs Mode* została zmieniona na wartość *Enable Kinematic Kinematic Pairs.* Model postaci miał już dodane te komponentu i nie trzeba było ich dodawać.



Rysunek 31. Diagram klasy ColliderController, opracowanie własne

Dla obiektów dłoni stworzony został także niewielki skrypt *ColliderController,* którego diagram został przedstawiony na rysunku 30. Listing 8 przedstawia kod funkcji *OnCollisionEnter*.

void OnCollisionEnter(Collision collision) {

GameObject hit = collision.gameObject;

if (hit.CompareTag("Head")) {

GameObject otherPlayer = hit.GetComponentInParent<Transform>().parent.parent.gameObject;

GameObject thisPlayer = transform.parent.parent.parent.gameObject;

if (thisPlayer != otherPlayer) {

otherPlayer.GetComponent<PlayerController>().CmdLose();

thisPlayer.GetComponent<PlayerController>().CmdWin();

}

}

}

Listing 8. Obsługa kolizji

Funkcja sprawdza, czy obiekt, z którym koliduje dłoń jest obiektem głowy. Jeśli uderzonym graczem jest inny gracz to wywoływane są funkcje stworzone w *PlayerController* – *CmdWin* i *CmdLose*. Są to funkcje serwerowe, każda z nich została oznaczona atrybutem *[Command]* i pozwalają na uruchamianie kodu tylko na serwerze. Aby wyświetlić komunikat na komputerach użytkowników potrzebne były funkcje klienckie *RpcWin* i *RpcLose,* obie oznaczone atrybutem *[ClientRpc]*. Takie funkcje uruchamiają się tylko na komputerach użytkowników, do których polecenie zostało wysłane.

Aby wyświetlić tekst, do sceny dodany został obiekt typu *Text.* Tworząc pierwszy element interfejsu użytkownika na scenie automatycznie pojawiły się także obiekty *Canvas*, w którym znajduje się cały interfejs użytkownika oraz *EventSystem*, który zajmuje się obsługą zdarzeń w Unity. Następnie do skryptu *PlayerController* dodane zostało wyszukiwanie obiektu tekstu podobnie, jak miało to miejsce wcześniej przy wyszukiwaniu obiektów dłoni i głowy. Mając do dyspozycji obiekt tekstu, w funkcjach *RpcWin* i *RpcLose* tekst został zmieniony tak, aby poinformować użytkownika  
o wygranej lub przegranej. Kod funkcji CmdWin i RpcWin został przedstawiony w listingu 9.

[Command]

public void CmdWin() {

if (isServer) {

RpcWin();

}

}

[ClientRpc]

public void RpcWin() {

if (isLocalPlayer) {

text.text = "You win!";

}

}

Listing 9. Funkcje odpowiadające za przekazanie użytkownikowi informacji o wygranej



Rysunek 32. Wygląd ekranu wygranej z perspektywy użytkownika, opracowanie własne

# TESTOWANIE

Do testów użyte zostały dwa komputery. Do jednego z nich został podłączony Oculus Rift wraz z kontrolerami. Drugi komputer pozwalał jedynie na obserwowanie  
i używanie interfejsu do stworzenia serwera lub połączenia się z istniejącym. Taka konfiguracja umożliwiła testowanie zarówno sytuacji gdy serwer stworzony jest na komputerze z Oculus Riftem, jak i bez niego.

Pierwszym testem było uruchomienie gry na obu komputerach i utworzenie serwera na komputerze z podłączonymi goglami i sprawdzenie, czy postać się porusza. Drugim, podobnym testem było uruchomienie gry na obu komputerach, ale w tym wypadku serwer został utworzony na komputerze bez podłączonego Oculus Rifta.

Kolejnymi testami testującymi zaimplementowaną funkcjonalność było sprawdzenie, czy stan gry jest zmieniany, po uderzeniu jednego gracza przez drugiego. Tak jak w przypadku testów przedstawionych wyżej, sprawdzone były przypadki gdy serwer jest uruchamiany na komputerze bez, i z goglami.

Testując, można było zauważyć, że występują niewielkie opóźnienia, rozwiązaniem tego problemu mogłoby być np. tworzenie delikatnego przejścia między kolejnymi pozycjami postaci, zamiast automatycznej zmiany pozycji. Ogólne odczucia w trakcie wolnej rozgrywki prezentowały się bardzo dobrze, opóźnienia niewiele przeszkadzały, a odzwierciedlenie ruchów dłoni było zadowalające. Problemy pojawiały się gdy użytkownicy poruszali się bardzo szybko, wtedy opóźniania były bardziej dotkliwe, a pozycje graczy nie zawsze odzwierciedlały ich faktyczną pozycję.

# PODSUMOWANIE

Celem pracy dyplomowej było ukazanie podejścia do rozwiązania problemu komunikacji sieciowej w grze umieszczonej w wirtualnej rzeczywistości. Cel pracy został osiągnięty. Aplikacja używa gogli wirtualnej rzeczywistości oraz realizuje przesyłanie położenia użytkownika poprzez wykorzystanie technologii sieciowej. Użytkownicy gry łączą się z serwerem. Stan gry oraz położenia obu graczy w przestrzeni jest zsynchronizowany.

Zostały ukazane technologie umożliwiające realizację komunikacji sieciowej oraz sposób przechwycenia i wysłania danych gracza używającego kontrolerów przechwytującego ruchy dłoni do innego gracza. Pokazana została specyfikacja gry oraz podejście zastosowane do jej budowania.

Aktualnie technologia wirtualnej rzeczywistości jest jedną z najprężniej rozwijających się w branży gier komputerowych. Przedstawiony sposób stworzenia gry w takim środowisku wraz z komunikacją między użytkownikami pozwala na zbudowanie środowiska, w którym uczucie immersji jest bardzo duże, jednak z uwagi na bliższy kontakt użytkownika z tworzoną aplikacją, wymagane jest większy nakład pracy niż w trakcie tradycyjnych gier.

# 



# BIBLIOGRAFIA

1. Unity User Manual (2018.2), <https://docs.unity3d.com/2018.2/Documentation/Manual>, (dostęp: 18.11.2018).
2. Oculus Documentation, <https://developer.oculus.com/documentation>,

(dostęp: 13.12.2018).

1. OpenVR API Documentation, <https://github.com/ValveSoftware/openvr/wiki/API-Documentation> (dostęp: 18.11.2018).
2. Unity – Multiplayer Networking, <https://unity3d.com/learn/tutorials/topics/multiplayer-networking> (dostęp 18.11.2018)
3. ​Gańko Tomasz, Gra-zabawka dla niemowląt przygotowana z użyciem w Unity 3D, Toruń 2017
4. Ioannou Nikolaos, Multi-Player​ ​Virtual​ ​Reality​ ​Game based​ ​on​ ​Hand​ ​Tracking​ ​Interaction, Chania 2017
5. Scrimger Rob, LaSalle Paul, Leitzke Clay, Parihar Mridula, Gupta Meeta, TCP/IP. Biblia, przeł. Jarczyk Adam, Gliwice, Wydawnictwo Helion, 2002

# SPIS ILUSTRACJI

[Rysunek 1. Główne okno edytora Unity, opracowanie własne 5](#_Toc535508094)

[Rysunek 2. Schemat Oculus Rifta 7](#_Toc535508095)

[Rysunek 3. Kontrolery Oculus Touch 8](#_Toc535508096)

[Rysunek 4. Okno przypisywania przycisków do akcji, opracowanie własne 9](#_Toc535508097)

[Rysunek 5. Okno główne Visual Studio 2018, opracowanie własne 10](#_Toc535508098)

[Rysunek 6. Schemat działania Git, opracowanie własne 11](#_Toc535508099)

[Rysunek 7. Schemat warstwy sieciowej w Unity, opracowanie własne 13](#_Toc535508100)

[Rysunek 8. Sześć stopni swobody 15](#_Toc535508101)

[Rysunek 9. Przykład śledzenia pozycji dla VR 16](#_Toc535508102)

[Rysunek 10. Ekran tworzenia projektu w Unity, opracowanie własne 19](#_Toc535508103)

[Rysunek 11. Okno konfiguracji opcji pluginu SteamVR, opracowanie własne 20](#_Toc535508104)

[Rysunek 12. Konfiguracji Opcji XR, opracowanie własne 21](#_Toc535508105)

[Rysunek 13. Okno edytora z uruchomioną podstawową sceną, opracowanie własne 22](#_Toc535508106)

[Rysunek 14. Resetowanie komponentu w Edytorze, opracowanie własne 22](#_Toc535508107)

[Rysunek 15. Usuwanie komponentu Box Collider, opracowanie własne 23](#_Toc535508108)

[Rysunek 16. Tworzenie nowego materiału, opracowanie własne 24](#_Toc535508109)

[Rysunek 17. Wygląd utworzonej mapy wraz z pokazaną hierarchią obiektów, opracowanie własne 25](#_Toc535508110)

[Rysunek 18. Konfiguracja komponentu szkieletu dla lewej dłoni, opracowanie własne 26](#_Toc535508111)

[Rysunek 19. Efekty wyboru opcji Range of Motion: Without Controller (po lewej) i With Controller (po prawej), opracowanie własne 27](#_Toc535508112)

[Rysunek 20. Nieprawidłowy (po lewej) i prawidłowy (po prawej) układ kości dłoni w modelu, opracowanie własne 27](#_Toc535508113)

[Rysunek 21. Dodawanie etykiet do projektu, opracowanie własne 28](#_Toc535508114)

[Rysunek 22. Schemat klasy PlayerController, opracowanie własne 29](#_Toc535508115)

[Rysunek 23. Konfiguracja komponentu Animator, opracowanie własne 32](#_Toc535508116)

[Rysunek 24. Wygląd maszyny stanów po dodaniu przejść między stanami, opracowanie własne 32](#_Toc535508117)

[Rysunek 25. Wybór animacji dla stanu Idle, opracowanie własne 33](#_Toc535508118)

[Rysunek 26. Diagram klasy IKHandling, opracowanie własne 34](#_Toc535508119)

[Rysunek 27. Wygląd postaci po zmianach w odwrotnej kinematyce, opracowanie własne 35](#_Toc535508120)

[Rysunek 28. Konfiguracja Network Managera, opracowanie własne 37](#_Toc535508121)

[Rysunek 29. Diagram klasy HandMovementCopier, opracowanie własne 38](#_Toc535508122)

[Rysunek 30. Konfiguracja skryptów sieciowych obiektu postaci, opracowanie własne 39](#_Toc535508123)

[Rysunek 31. Diagram klasy ColliderController, opracowanie własne 40](#_Toc535508124)

[Rysunek 32. Wygląd ekranu wygranej z perspektywy użytkownika, opracowanie własne 41](#_Toc535508125)

1. *Unity Asset Store, https://assetstore.unity.com/* [↑](#footnote-ref-1)
2. Schemat pochodzi z *Oculus Rift Development Kit 2 Quick Start Guide* [↑](#footnote-ref-2)
3. Immersja – proces zanurzania się użytkownika w rzeczywistość elektroniczną [↑](#footnote-ref-3)
4. Schemat pochodzi z *Oculus Touch, la manette VR, sera disponible le 6 décembre – Geeko*, <https://geeko.lesoir.be/2016/10/07/oculus-touch-la-manette-vr-sera-disponible-le-6-decembre/> (dostęp 09.01.2019) [↑](#footnote-ref-4)
5. Asset jest reprezentacją dowolnej rzeczy np. modelu, dźwięku, skryptu lub zestawu skryptów w projekcie. *Unity – Manual: Asset Workflow*, <https://docs.unity3d.com/Manual/AssetWorkflow.html>(dostęp 12.01.2019) [↑](#footnote-ref-5)
6. Paralaksa to efekt, który sprawia, że obiekty znajdujące się daleko od obserwującego będącego w ruchu wydają się przesuwać wolniej od tych blisko obserwującego [↑](#footnote-ref-6)
7. *Six degrees of freedom – Wikipedia*, https://en.wikipedia.org/wiki/Six\_degrees\_of\_freedom (dostęp: 12.01.2019) [↑](#footnote-ref-7)
8. *How does Oculus Quest work? – Quora*, https://www.quora.com/How-does-the-Oculus-Quest-work (dostęp 12.01.2019) [↑](#footnote-ref-8)
9. Sci Fi Officer Captain w Unity Asset Store, https://assetstore.unity.com/packages/3d/characters/sci-fi-officer-captain-52376 (dostęp: 19.11.2018) [↑](#footnote-ref-9)
10. *Unity - Scripting API: MonoBehaviour*, https://docs.unity3d.com/ScriptReference/MonoBehaviour.html (dostęp 15.0.2019) [↑](#footnote-ref-10)
11. Kwaterniony – struktura algebraiczna będąca rozszerzeniem ciała liczb zespolonych, https://pl.wikipedia.org/wiki/Kwaterniony (dostęp 10.01.2019) [↑](#footnote-ref-11)
12. *Unity – Scripting API: Quaternion*, https://docs.unity3d.com/ScriptReference/Quaternion.html (dostęp: 10.01.2019) [↑](#footnote-ref-12)
13. Prefab jest obiektem zapisanym w plikach projektu i który działa jako swego rodzaju szablon dla innych obiektów. Kopie prefabów można dynamicznie tworzyć w aplikacji, *Unity – Manual: Prefabs,* <https://docs.unity3d.com/Manual/Prefabs.html> (dostęp 12.01.2019) [↑](#footnote-ref-13)